A Type-Directed Abstraction Refinement Approach to Higher-Order Model Checking

Steven J. Ramsay
University of Oxford
steven.ramsay@cs.ox.ac.uk

Robin P. Neatherway
University of Oxford
robin.neatherway@cs.ox.ac.uk

C.-H. Luke Ong
University of Oxford
luke.ong@cs.ox.ac.uk

Abstract

The trivial-automaton model checking problem for higher-order recursion schemes has become a widely studied object in connection with the automatic verification of higher-order programs. The problem is formidably hard[1]: despite considerable progress in recent years, no decision procedures have been demonstrated to scale robustly beyond recursion schemes that comprise more than a few hundred rewrite rules. We present a new, fixed-parameter polynomial time algorithm, based on a novel, type directed form of abstraction refinement in which behaviours of a scheme are distinguished by the abstraction according to the intersection types that they inhabit (the properties that they satisfy). Unlike other intersection type approaches, our algorithm reasons both about acceptance by the property automaton and acceptance by its dual, simultaneously, in order to minimize the amount of work done by converging on the solution to a problem instance from both sides. We have constructed PREFACE, a prototype implementation of the algorithm, and assembled an extensive body of evidence to demonstrate empirically that the algorithm readily scales to recursion schemes of several thousand rules, well beyond the capabilities of current state-of-the-art higher-order model checkers.

Categories and Subject Descriptors F [*3*]: 1; D [*2*]: 4

Keywords higher-order model checking; intersection types; abstraction refinement

1. Introduction

Higher-order model checking, or the model checking problem for trees generated by higher-order recursion schemes (HORS), is a widely studied decision problem in connection with the theory and practice of the verification of higher-order programs. Since HORS are simultaneously very expressive [21], algorithmically well-behaved [20], and able to accurately model higher-order control flow [8], they are an appealing target for algorithmic verification procedures for functional programs [12, 16, 17, 22, 26]. Indeed, in a precise sense, HORS are the higher-order analogue of Boolean programs, which have played a very successful rôle in the verification of first order, imperative programs [1].

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535873

It is for these reasons, and despite the severe worst-case complexity of the problem[1], that several ingenious algorithms [3, 4, 12, 13, 19] have recently been developed with the aim of solving the higher-order model checking problem for many "practical" instances. However, the state of this effort is summarised well by the authors of [3]:

> "The state-of-the-art model checker TRECS [12] can handle a few hundred lines of HORS generated from various program verification problems. It is, however, not scalable enough to support automated verification of thousands or millions of lines of code. Thus, obtaining a better higher-order model checker is a grand challenge in the field..."

Our main contribution is a new algorithm for higher-order model checking and a large body of evidence to show empirically that it scales well to HORS consisting of several thousand rules. In contrast, the largest instances considered in the literature to date are of the order of several hundred rules. By way of an example, the order-2 benchmark $\mathcal{G}_{2,10000}$ of Kobayashi [12], which consists of 10006 rules, can be processed by our prototype implementation in less than one minute.

Our algorithm, which decides the HORS model checking problem with respect to alternating trivial tree automata, has been designed to be scalable. Since the inherent worst-case complexity of HORS model checking is extreme, to have any chance at all of solving non-trivial instances, one has to work in the belief that those instances that are met in practice are not pathological. Hence, it is essential to ensure that only work that is *relevant* to deciding the particular instance at hand is actually computed. To help achieve this goal, our algorithm is designed in the abstraction refinement paradigm [6]. Initially a relatively cheap but coarse-grained approximation to the problem is processed and, as much as possible, detail is only added by successive iterations where the problem instance necessitates it. Moreover, it can be shown that our algorithm is *fixed-parameter polynomial time* in the size of the scheme; the parameters that are fixed are the order and arity of the scheme, and the size of the tree automaton.

Our algorithm exploits the characterisation of higher-order model checking as an intersection type inference problem [11, 14], representing the state of knowledge about the behaviours of the recursion scheme as a pair of type environments, called the *context*, which assigns intersection types to the non-terminals of the scheme. As the algorithm progresses, the number of types (and hence state of knowledge) in the environments increases, until after some finite number of iterations there will be enough type information to decide the property one way or the other. Furthermore, this limit context will form a certificate of the decision that is independently verifiable by intersection type checking.

[1] n-EXPTIME complete for recursion schemes of order n [15, 20]

In order to gain more information and thus populate the context, each iteration consists of constructing a sound abstraction of the configuration graph [12] of the scheme. Since recursion schemes have no facility to inspect the data that they operate over, the behaviours of the scheme arise from the complex interactions between higher-order functions. Hence, we have designed this abstraction around a traditional CFA [10], but with an important twist: in our abstraction, parameters to function calls are distinguished according to the intersection types that they inhabit, in other words, according to the properties that they satisfy. This is, in turn, a function of the context and hence, as the algorithm progresses and the size of the context increases, so the abstractions become more precise, as they are able to distinguish more instances of function calls.

Such an abstract configuration graph is a concise but approximate representation of all the possible reduction sequences of the scheme. Through its analysis, the algorithm can classify certain behaviours that can be seen to generate trees that are accepted by the property automaton and certain other behaviours that can be seen to generate trees that are rejected by the property automaton. From the former it is able to extract new "acceptance" types and from the latter new "rejection" types and both are added to the context ready to proceed with the next iteration. Indeed, a key feature of the algorithm, and a novelty among intersection type based decision procedures, is that it uses types to reason both about property automaton acceptance *and* rejection, simultaneously.

We have implemented the algorithm in a tool, PREFACE, and evaluated its performance over the several hundred problem instances that are now either recorded in the literature or which have resulted from verification tools for higher-order programs. These instances range from a few tens of rules to several thousands and from first order schemes up to fifth order, and a few beyond. The results show very clearly that, whilst PREFACE is sometimes a little slower than other model checkers on examples up to around one hundred rules, its great strength is in solving examples of many hundreds of rules, where it performs consistently better than other model checkers, and several thousands of rules, which are typically instances that it alone can solve.

Outline The rest of the article is structured as follows. In section 2 we fix notation and preliminary definitions. In section 3 we give an informal outline of the algorithm by means of an example. In section 4 the algorithm is defined formally. In section 5 we discuss our prototype implementation and present a digest of the empirical evaluation and associated analysis. In section 6 we discuss related work. All proofs of claims in the text are relegated to the appendix of the long version of this work [24], which also includes a guided run of the algorithm on a second example instance and a full transcript of the empirical evaluation.

2. Preliminaries

We assume throughout a denumerable set $(F,\ G,\ H\ \in)\ \mathcal{F}$ of *function symbols* and a disjoint, denumerable set $(x,\ y,\ z\ \in)\ \mathcal{V}$ of *variables*.

Labelled trees Let A be a set without restriction. An *A-labelled tree* is a partial function $T : \mathbb{N}^* \to A$ whose domain is prefix closed. In case the set A is *ranked*, that is, each symbol $a \in A$ has a specified arity $\mathsf{arity}(a) \in \mathbb{N}$, then the tree T can be said to be *well-ranked* just if, whenever $T(w) = a$ and $\mathsf{arity}(a) = n$ then: $w \cdot i \in \mathsf{dom}(T)$ iff $i \in [1..n]$.

Simple kinds. The *simple kinds*[2] over the kind of trees o, denoted $(\kappa\ \in)\ \mathbb{S}$, are formed by the grammar $\kappa\ ::=\ o\ \mid\ \kappa_1\ \to\ \kappa_2$. As usual, we use parentheses to disambiguate the structure of such expressions, observing that the arrow associates to the right. The *arity* and *order* of a simple kind are natural numbers defined as usual. If a simple kind has order 0 (and hence has arity 0) we say that it is *ground*.

Raw terms. Let $(a,\ b,\ c\ \in)\ \Sigma$ be a set of atomic constants. The set of *raw terms over* Σ, denoted $(s,\ t,\ u,\ v\ \in)\ T_\Sigma(\mathcal{F},\ \mathcal{V})$, is defined by the grammar:

$$s,\ t\quad ::=\quad x\ \mid\ F\ \mid\ c\ \mid\ s\ t$$

The *free variables* of a term t, denoted $\mathsf{FV}(t)$, is just the set of variables that occur in t. A term t with $\mathsf{FV}(t)$ empty is called *closed* and the set of all closed terms is denoted $T_\Sigma(\mathcal{F})$. We denote the set of closed terms which, moreover, contain no occurrences of function symbols by T_Σ. In case the atomic constants are said to be *kinded* we assert that there is an associated kinding function kind which maps each constant $c \in \Sigma$ to a first-order kind in $\mathbb{S}$.

Kinded terms. A kind environment Δ, is a finite, partial function from $\mathcal{V} \cup \mathcal{F}$ to $\mathbb{S}$. A *kind judgement* is an expression of the form $\Delta \vdash t : \kappa$. We omit the standard definition of kind assignment to terms, i.e. simple type assignment.

Recursion Schemes. A higher-order recursion scheme (HORS) $\mathcal{G}$ is a tuple $\langle \Sigma,\ \mathcal{N},\ \mathcal{R},\ S\rangle$ in which:

- The alphabet of *terminal symbols* $(a,\ b,\ c\ \in)\ \Sigma$ is a finite set of first-order, kinded constants.
- The alphabet of *non-terminal symbols*, $(F,\ G,\ H\ \ \in)\ \mathcal{N}$, is a finite set of kinded function symbols, disjoint from Σ. We will sometimes view $\mathcal{N}$ as a kind environment mapping non-terminals $F \in \mathcal{N}$ to their kinds $\mathsf{kind}(F)$.
- The *rewrite rules*, $\mathcal{R}$, comprise a function mapping each non-terminal symbol F of kind $\kappa_1 \to \cdots \to \kappa_n \to o$ to an expression $\lambda x_1\ \cdots\ x_n.\ t$, such that:

  $$x_1 : \kappa_1,\ \cdots,\ x_n : \kappa_n \vdash t : o$$

  is a provable assignment of kinds to terms. We will often write $(F,\ \lambda x_1, \ldots, x_n.\ t) \in \mathcal{R}$ as an equation $F = \lambda x_1, \ldots, x_n.\ t$.
- The *start symbol*, $S \in \mathcal{N}$, is a non-terminal symbol of kind o.

Each recursion scheme is assigned an *order* which is given by the maximum order of (the kind of) its non-terminal symbols. Recursion schemes have a simple notion of reduction, which is defined as the contextual closure of the following rule:

$$\frac{\mathcal{R}(F) = \lambda x_1\ \cdots\ x_n.\ t}{F\ s_1\ \cdots\ s_n \Rightarrow t[s_1/x_1, \ldots, s_n/x_n]}$$

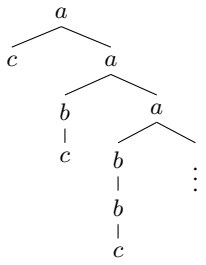
For the purposes of model checking, we are interested in the trees generated by the scheme. The *value tree* of a scheme $\mathcal{G}$, denoted $\mathsf{Tree}(\mathcal{G})$ is the (possibly infinite) term tree obtained by reducing the start symbol *ad infinitum*. To account for the possibility of infinite, unproductive recursion, the value tree is defined as follows. First, introduce a new symbol of zero arity, $\bot$, into Σ and consider the least preordering of Σ that asserts $\bot\ \leq\ a$ for all $a\ \in\ \Sigma$. Next, for each closed term t of ground type, define $t^\bot$ recursively by: (i) $(F\ s_1\ \cdots\ s_n)^\bot = \bot$ and (ii) $(a\ s_1\ \cdots\ s_n)^\bot = a\ s_1^\bot\ \cdots\ s_n^\bot$. This mapping sends each term to well-ranked tree in the complete partial order of $\Sigma \cup \bot$-labelled trees, which are ordered by letting $T_1 \sqsubseteq T_2$ just if, for all $\pi \in \mathsf{dom}(T_1)$, $T_1(\pi) \leq T_2(\pi)$. Finally, we set $\mathsf{Tree}(\mathcal{G}) = \bigsqcup\{t^\bot \mid S \Rightarrow^* t\}$.

Example 1. Consider the first-order scheme over terminal symbols $a : o \to o \to o$, $b : o \to o$ and $c : o$ in which the non-terminal symbols $S : o$ and $F : o \to o$ are defined by the equations:

[2] These are nothing more than the simple types over the base type o, but we prefer to use the word *kind* to avoid conflict with intersection *type* later.

$$\begin{array}{lcl} S & = & F\ c \\ F & = & \lambda x.\ a\ x\ (F\ (b\ x)) \end{array}$$

By reducing *ad infinitum*, the start symbol S generates an infinite a, b and c-labelled tree, a prefix of which is depicted on the right. This tree has no $\bot$-labelled nodes since every redex contraction produces a new terminal symbol in head position.



Positive Boolean formulae. Given a finite set X, the *positive Boolean formulas over* X, denoted $(\phi \in) \mathsf{B}^+(X)$, are defined by the grammar $\phi ::= \mathsf{t} \mid \mathsf{f} \mid x \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$. Given a positive Boolean formula ϕ, an *assignment* is a finite subset S of X. An assignment S is said to be a *satisfying assignment* for ϕ, written $S \models \phi$, when assigning t to elements of S and f to elements of $X \setminus S$ makes ϕ true.

Alternating (co-)trivial tree automata. An alternating (co-)trivial tree automaton (ATT) $\mathcal{A}$ is a tuple $\langle \Sigma,\ Q,\ \delta,\ q_0,\ F \rangle$ in which Σ, is a finite set of ranked constants, $(q \in) Q$, is a finite set of *states*, the *transition function*, δ, is a function in $\Pi_{(q,a) \in Q \times \Sigma} \cdot \mathsf{B}^+([1..\mathsf{arity}(a)] \times Q)$, the *initial state* is $q_0 \in Q$ and the *accepting states*, F, are either all of Q or empty. In case $F = Q$, we say that the ATT has a *trivial* acceptance condition, otherwise $F = \emptyset$ and we say that it has a *co-trivial* acceptance condition. More often than not we will simply introduce a given automaton as a trivial or a co-trivial automaton and omit the final component. Furthermore, when specifying particular automata, we will elide clauses of the transition function whose image is f.

Given a Σ-ranked and labelled tree T, a *run tree of* $\mathcal{A}$ *on* T is a $(\mathsf{dom}(T) \times Q)$-labelled, unranked tree R satisfying:

(APT-1) $R(\epsilon) = (\epsilon,\ q_0)$

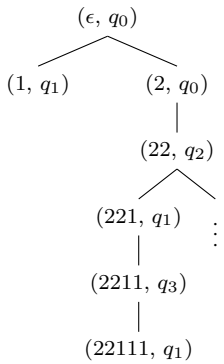
(APT-2) For all $w \in \mathbb{N}^*$, if $R(w) = (w',\ q)$ then there is some set S that satisfies $\delta(q,\ T(w'))$ and, for all $(i,\ q') \in S$, there exists some $j \in \mathbb{N}$ such that $R(w \cdot j) = (w' \cdot i,\ q')$.

We say that a run tree R is *accepting* just if, on every infinite branch of R, there is some state $q \in F$ which occurs infinitely often. The *language* of an ATT $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of Σ-ranked and labelled trees T for which there exists an accepting run-tree on T. We define the *complement* of $\mathcal{A}$, denoted $\mathcal{A}^c$, by the standard de Morgan dual construction, which ensures that $\mathcal{L}(\mathcal{A})^c = \mathcal{L}(\mathcal{A}^c)$ [18]. Note, the dual of an ATT with a trivial acceptance condition is an ATT with a co-trivial acceptance condition.

Example 2. Consider the ATT over the states q_0, q_1, q_2 and q_3, in which the transition function is defined by the following clauses:

$$\begin{array}{lcl} \delta(q_0,\ a) & = & ((1,\ q_1) \wedge (2,\ q_0)) \vee (2,\ q_2) \\ \delta(q_1,\ b) & = & (1,\ q_3) \\ \delta(q_1,\ c) & = & \mathsf{t} \\ \delta(q_2,\ a) & = & (1,\ q_1) \wedge (2,\ q_0) \\ \delta(q_3,\ b) & = & (1,\ q_1) \end{array}$$

This ATT accepts those trees that have an infinite a-labelled spine and of every two consecutive branches off the spine, at least one is required to be labelled by an even number of b nodes terminated by a c. A prefix of the run tree over the tree generated by the scheme in Example 1 is depicted to the right.



Higher-order model checking. We define $\mathcal{A}^\bot$ as the ATT $\mathcal{A}$ augmented with extra transitions so as to accept the symbol $\bot$ from every state. The ATT-model checking problem for HORS is, given a HORS $\mathcal{G}$ and an ATT $\mathcal{A}$, to determine the truth of the assertion $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\bot)$.

Intersection types. In what follows fix an ATT $\mathcal{A}$. We consider *intersection types* [7]. As is usual in the higher-order model checking literature, we make a distinction between strict types and intersection types (borrowing the terminology from van Bakel [27]). The *intersection types* over $\mathcal{A}$, denoted $\mathbb{I}_\mathcal{A}$, are defined simultaneously with the *strict types* over $\mathcal{A}$ by the following grammar:

$$\begin{array}{rrcl} \text{(Strict Types)} & \tau & ::= & q \mid \sigma \to \tau \\ \text{(Intersection Types)} & \sigma & ::= & \bigwedge_{i=1}^{n} \tau_i \end{array}$$

in which $q \in Q$ and $n \geq 0$. We will use τ and σ to denote strict and intersection types respectively. When we are agnostic about whether a particular expression is either a strict type or an intersection type we will say it is simply a *type* and denote it by θ. When no confusion can arise, we will typically write $\top$ for the empty intersection, an intersection containing two elements infix as $\tau_1 \wedge \tau_2$ and an intersection of the singleton set containing τ simply as τ. Given an intersection $\sigma = \bigwedge_{i=1}^{n} \tau_i$ we will often identify σ with its set of conjuncts $\{\tau_1,\ \ldots,\ \tau_n\}$, writing assertions such as $\tau \in \sigma$ and $\sigma_1 \subseteq \sigma_2$ with the obvious interpretation. Finally, we shall have no qualms about constructing the intersection of intersection types, since this can be given naturally as the intersection of the union over their respective strict conjuncts.

Intersection subtyping. There is a natural subtype preorder on intersection types, which was first explicitly considered by Barendregt, Coppo and Dezani-Ciancaglini in [2]. We shall use the following variant, defined inductively by the following clauses.

(Q-Bas) $q \leq q$

(Q-Arr) if $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$ then $\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$

(Q-Prj) for all $i \in [1..n]$, $\bigwedge_{j=1}^{n} \tau_j \leq \tau_i$

(Q-Glb) if, for all $i \in [1..n]$, $\sigma \leq \tau_i$, then $\sigma \leq \bigwedge_{j=1}^{n} \tau_j$

(Q-Trs) if $\theta_1 \leq \theta_2$ and $\theta_2 \leq \theta_3$ then $\theta_1 \leq \theta_3$

Intersection type environment. An *intersection type environment* Γ is a finite, partial function from $\mathcal{F} \cup \mathcal{V}$ to $\mathbb{I}_\mathcal{A}$. We will often view type environments as total functions assigning $\Gamma(F) = \top$ whenever $F \notin \mathsf{dom}(\Gamma)$. We will write $\Gamma_1 \uplus \Gamma_2$ for the operation sometimes called *type environment multiplication*, which is just the pointwise combination of environments defined by:

$$(\Gamma_1 \uplus \Gamma_2)(F) = \Gamma_1(F) \wedge \Gamma_2(F)$$

and write $\Gamma_1 \Subset \Gamma_2$ just if there is some Γ' and $\Gamma_1 \uplus \Gamma' = \Gamma_2$. We will write $\Gamma \upharpoonright X$ for the restriction of Γ to only those typings whose subject lies in X. Finally, we also extend the subtype relation to environments pointwise, writing $\Gamma_1 \leq \Gamma_2$ just if, $\mathsf{dom}(\Gamma_2) \subseteq \mathsf{dom}(\Gamma_1)$ and, for all $\xi \in \mathsf{dom}(\Gamma_2)$, $\Gamma_1(\xi) \leq \Gamma_2(\xi)$.

Intersection type assignment. An *intersection type judgement* is an expression of the form $\Gamma \vdash t : \tau$ (with τ a strict type) whose derivations are defined inductively by the system in Figure 1. Note that in that system, we use the notation $S|_i$ to denote the set $\{q \mid (i,\ q) \in S\}$. Given a type environment Γ and a term t, we define the set of all strict types assignable to t under Γ by:

$$\mathbb{T}(\Gamma)(t) = \{\tau \mid \Gamma \vdash t : \tau\}$$

The type system is induced by the property automaton $\mathcal{A}$, which features in the premise to the rule (T-Cst). This rule acts to give a meaning to intersection types that can be thought of as follows. Each base type q is the type of all terms t that generate (via infinite

$$\frac{}{\Gamma,\, x : \bigwedge_{i=1}^{n} \tau_i \vdash x : \tau_i} \text{ (T-Var)}$$

$$\frac{}{\Gamma,\, F : \bigwedge_{i=1}^{n} \tau_i \vdash F : \tau_i} \text{ (T-Fun)}$$

$$\frac{S \models \delta(q,\, c)}{\Gamma \vdash c : \bigwedge(S|_1) \to \cdots \to \bigwedge(S|_n) \to q} \text{ (T-Cst)}$$

$$\frac{\Gamma \vdash s : \sigma \to \tau \qquad \Gamma \vdash t : \tau' \ [\forall \tau' \in \sigma'] \qquad \sigma' \leq \sigma}{\Gamma \vdash s\ t : \tau} \text{ (T-App)}$$

Figure 1. Assignment of types to terms.

$$\frac{}{q :: o} \text{ (K-Bas)}$$

$$\frac{\sigma :: \kappa_1 \qquad \tau :: \kappa_2}{\sigma \to \tau :: \kappa_1 \to \kappa_2} \text{ (K-Arr)}$$

$$\frac{\tau_i :: \kappa \ (i \in [1..n])}{\bigwedge_{i=1}^{n} \tau_i :: \kappa} \text{ (K-Int)}$$

Figure 2. Assignment of kinds to types.

reduction) trees that are accepted by the automaton from state q. An intersection such as $q_1 \wedge q_2$, is the type of all terms that generate trees that are accepted *both* from state q_1 and from state q_2. Finally, an arrow such as $q_1 \wedge q_2 \to q$ is the type of those terms which, when applied to a term that generates a tree accepted from q_1 and q_2, will, as an application, generate a tree accepted from state q.

In this work we will be concerned both with type assignment in the intersection type system induced by property automaton $\mathcal{A}$ and type assignment in the intersection type system induced by the dual of this automaton $\mathcal{A}^c$. Hence, whenever needed we will try to disambiguate which notion of type assignment we are referring to by annotating the notation with $\mathcal{A}$ or $\mathcal{A}^c$.

Intersection refinement types. The *intersection refinement types* over Q are those types θ for which there is some kind κ such that $\theta :: \kappa$, pronounced "σ refines κ", is provable in the system of kind assignment below. The *strict refinement types* over Q are defined as the obvious restriction of this system. We lift the refinement relation to environments by writing $\Gamma :: \Delta$ just if, for all $F : \sigma \in \Gamma$, there is a typing $F : \kappa \in \Delta$ and $\sigma :: \kappa$.

Type environment consistency. We say that an intersection type environment Γ is ($\mathcal{G}$,$\mathcal{A}$)-*consistent* just if, for each typing $F : \sigma \in \Gamma$ such that $F \in \mathsf{dom}(\mathcal{N})$, there is a *possibly infinite* witness, rooted at $\Gamma \rhd F : \sigma$, and built according to the following system:

$$\frac{\mathcal{R}(F) = \lambda x_1 \ldots x_n.\, t \qquad \Gamma,\, x_1 : \sigma_1, \ldots, x_1 : \sigma_n \vdash t : q \qquad \Gamma \rhd G : \sigma\ (\forall G : \sigma \in \Gamma)}{\Gamma \rhd F : \sigma_1 \to \cdots \to \sigma_n \to q} \qquad \frac{\Gamma_1 \trianglelefteq \Gamma \quad \Gamma_1 \rhd F : \tau_1 \quad \cdots \quad \Gamma_n \trianglelefteq \Gamma \quad \Gamma_1 \rhd F : \tau_n}{\Gamma \rhd F : \bigwedge_{i=1}^{n} \tau_i}$$

Similarly, we say that an intersection type environment Γ is ($\mathcal{G}$,$\mathcal{A}$)-*co-consistent* just if, for each typing $F : \sigma \in \Gamma$ there is a *strictly finite* witness built from the above system. The next theorem follows from Kobayashi and Ong [14].

Theorem 1. *Fix a scheme $\mathcal{G}$ and ATT $\mathcal{A}$.*

(i) $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ *iff there exists* $(\mathcal{G},\, \mathcal{A})$*-consistent* $\Gamma :: \mathcal{N}$ *and* $q_0 \in \Gamma(S)$.

(ii) $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}((\mathcal{A}^{\perp})^c)$ *iff there exists* $(\mathcal{G},\, \mathcal{A}^c)$*-co-consistent* $\Gamma :: \mathcal{N}$ *and* $q_0 \in \Gamma(S)$.

3. Type directed abstraction refinement

The starting point for the algorithm is the characterisation of the trivial automaton model checking problem for recursion schemes given in Theorem 1. Our algorithm tries to prove (a) $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ and (b) $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}((\mathcal{A}^{\perp})^{\mathsf{c}})$ simultaneously, by iteratively constructing two type environments $\Gamma_\exists$ and $\Gamma_\forall$ which are possible witnesses to (a) and (b) respectively. Since an invariant of the algorithm is that $\Gamma_\exists$ is always ($\mathcal{G}$, $\mathcal{A}$)-consistent and $\Gamma_\forall$ is always ($\mathcal{G}$, $\mathcal{A}^{\mathsf{c}}$)-co-consistent, it follows that at most one of the two environments can prove the type assignment $S : q_0$; in fact, upon termination, exactly one of the two will do so. Since the two type systems $\mathcal{A}$ and $\mathcal{A}^{\mathsf{c}}$ share the same underlying set of types, let us call an intersection type an *acceptance type* when we regard it as part of the system $\mathcal{A}$ and let us call it a *rejection type* when we regard it as part of the system $\mathcal{A}^{\mathsf{c}}$.

The algorithm starts with $\Gamma^0_\exists = \Gamma^0_\forall = \emptyset$, which is trivially (co-)consistent. On each iteration, new type assignments are inferred which will be added to one or the other of the environments. The way that these new types are inferred is by, on each iteration, constructing and interrogating an auxiliary structure, called the *abstract configuration graph*. As its name suggests, this graph is an abstraction and the precision of the abstraction is a function of the size of the type environments $\Gamma_\exists$ and $\Gamma_\forall$. The more precise the abstraction the more useful the type information that can be deduced by interrogating it. Hence, starting from the empty *context*, $C_0 = \langle \Gamma^0_\exists,\, \Gamma^0_\forall \rangle$, the abstraction refinement cycle continues as follows. In iteration $i+1$ with context $C_i = \langle \Gamma^i_\exists,\, \Gamma^i_\forall \rangle$, the abstract configuration graph is constructed. Two subgraphs are then carved out called the accepting and rejecting regions. These regions are the parts of the graph from which it is possible to obtain consistent acceptance typings and co-consistent rejection typings respectively. Types are then extracted from the regions and added to context C_i to form a strictly larger context $C_{i+1} = \langle \Gamma^{i+1}_\exists,\, \Gamma^{i+1}_\forall \rangle$. If one of these two environments can already type $S : q_0$ then the algorithm terminates. Otherwise the cycle repeats and, since C_{i+1} is strictly larger, the abstract configuration graph constructed in iteration $i+2$ will be strictly more precise, and new type information will be deduced. Since there are only finitely many intersection refinement types associated with a given scheme one of the environments will be eventually become saturated and thus witness the corresponding assertion (a) or (b).

The configuration graph of a model checking problem instance was introduced by Kobayashi and Ong in [12, 14]. It is a kind of product construction, pairing up the reduction relation of the scheme and the transition function of the property automaton and it takes the shape of a rooted, directed graph. Since we are solving the alternating trivial automaton model checking problem for recursion schemes, the configuration graphs that we are interested in are something in between the simple ones of [12], which are appropriate to deterministic trivial automaton properties, and the much more complicated ones of [14], which are appropriate to alternating parity automaton properties. Moreover, we shall not be interested in the configuration graphs themselves, which are in general infinite, but rather in finite abstractions.

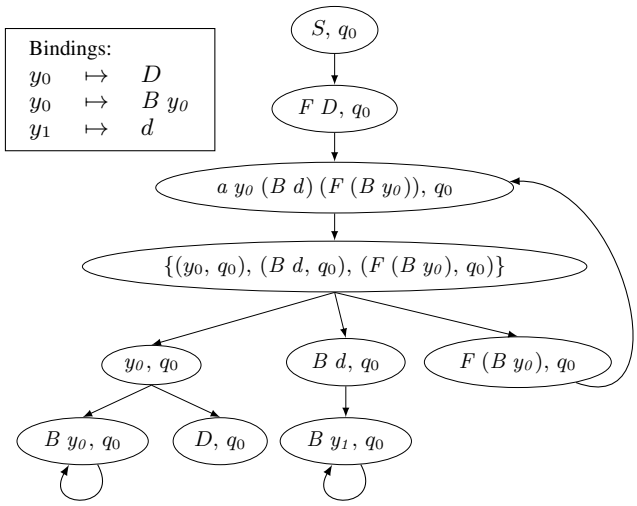


Figure 3. An abstract configuration graph.

Consider the following model checking instance, consisting of recursion scheme $\mathcal{G}$ over the terminal symbols a and d and a two-state trivial automaton $\mathcal{A}$:

$$\begin{array}{lcl} S & = & F\ D \\ F & = & \lambda x.\, a\ x\ (B\ d)\ (F\ (B\ x)) \\ B & = & \lambda z.\, B\ z \\ D & = & d \end{array} \qquad \begin{array}{lcl} \delta(q_1,\, d) & = & \mathsf{t} \\ \delta(q_0,\, a) & = & (1,\, q_0) \\ & & \wedge (2,\, q_0) \\ & & \wedge (3,\, q_0) \end{array}$$

As described above, the algorithm begins with the initial context $C_0 = \langle \Gamma^0_\exists,\, \Gamma^0_\forall \rangle$ consisting of $\Gamma^0_\exists = \emptyset$ and $\Gamma^0_\forall = \emptyset$. In order to infer new type assignments it constructs the abstract configuration graph $\mathsf{ACG}(C_0)$, which is given in full in Figure 3. The construction of the graph starts at the root, which is a vertex labelled $(S,\, q_0)$. Every vertex of the graph is either labelled by a pair $(t,\, q)$ of a ground kind term and a state, called a *configuration*, or by a finite set of such configurations. If there is a vertex $(t,\, q)$ in a graph $\mathsf{ACG}(\langle \Gamma_\exists,\, \Gamma_\forall \rangle)$, then it should be read as "neither $\Gamma_\exists \vdash_{\mathcal{A}} t : q$ nor $\Gamma_\forall \vdash_{\mathcal{A}^c} t : q$ is provable". Hence, the starting point for the graph $\mathsf{ACG}(C_0)$ is the fact that $S : q_0$ is not provable in $\emptyset$.

The shape of the successors of a vertex, if it has any, depend upon whether the vertex is a configuration or a set and in case of the former, the syntactic class of which its head symbol is a member. In this case the root of the graph is a configuration and its head symbol is S which is a non-terminal. Hence the term part of this configuration is a redex. In such cases, the vertex has at most one successor and that successor *represents* the contraction of the redex. The contraction of the redex S is $F\ D$ and, because neither $\Gamma^0_\exists \vdash_{\mathcal{A}} F\ D : q_0$ nor $\Gamma^0_\forall \vdash_{\mathcal{A}^c} F\ D : q_0$ is provable, it has a successor, which is labelled by $(F\ D,\, q_0)$. Since this vertex is of the same form, it also has a single successor which represents the contraction of the redex, however the situation is more complicated because this redex involves parameters. It is along such edges that the abstraction happens: rather than substituting actual parameters for formals in the successor, formals are substituted by special variables which are used by the abstraction to represent sets of terms. So we create a new variable y_0 and make a note $y_0 \mapsto D$ that one of the possible instantiations of y_0 is D. In this way, we are always able to recover the real contraction $a\ D\ (B\ d)\ (F\ (B\ D))$ from the abstract one $a y_0 (B d) (F (B y_0))$ by rewriting occurrences of the variables y_0 using the *bindings* $y_0 \mapsto D$. This ensures we are building a *sound* abstraction, in the sense of incorporating all the behaviours of the original. This kind of abstraction is, in essence, a traditional control flow analysis [10], but here we go one step further, which is critical in order to obtain completeness. The extra step that we take is to record the acceptance type and rejection type of the new variable y_0. The acceptance type (respectively rejection type) of y_0 is just the intersection of types assignable to the term that it has been introduced to represent, in other words, $\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma^0_\exists)(D) = \top$ (respectively $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma^0_\forall)(D) = \top$). The types of such variables will later determine when new variables should be created in order to represent actual parameters, as y_0 was here, or if variables created previously should be reused.

The frontier of our construction so far consists of a terminal headed configuration. The successors of a configuration of the form $(a\ s_1 \cdots s_n,\, q)$ depend upon the satisfying assignments to $\delta(q,\, a)$. For each satisfying assignment S there is one successor, which is the set of configurations $\{(s_i,\, q') \mid (i,\, q') \in S\}$. In this case, $\delta(q,\, a)$ is satisfied just if the first argument of a is accepted from q_0, the second argument is accepted from q_0 and the third argument is accepted from q_0. Consequently, the set is $\{(y_0,\, q_0),\, (B\ d,\, q_0),\, (F\ (B\ y_0),\, q_0)\}$. Such a set $\{(t_1,\, q_1), \ldots, (t_n,\, q_n)\}$ should be read as "there is some i such that $\Gamma_\exists \vdash_{\mathcal{A}} t_i : q_i$ is not provable". The successors of the set are those configurations $(t,\, q)$ in the set which are not provable in either $\Gamma_\exists$ or $\Gamma_\forall$.

So let us now consider the frontier configuration $(F(B\,y_0),\, q_0)$. Since it is a non-terminal headed configuration, if it has a successor then the successor abstractly represents the contraction in the same way as before. However, since the actual parameter $B\ y_0$ has the same acceptance type and rejection type as the previous one that we considered, D (and is the same kind), we will not create a new variable to represent $B\ y_0$, but we will *reuse* the variable y_0 that we used to represent actual parameter D. So the abstract contraction is the term $a\ y_0\ (B\ d)\ (F\ (B\ y_0))$ and we form a loop in the graph. To ensure the abstraction remains sound we note that another possible instantiation of y_0 is $B\ y_0$, but observe that this merging of vertices resulting from reusing y_0 has caused some loss of information. Now the term $a\,y_0\,(B\,d)\,(F(B\,y_0))$ represents many different concrete instances, including some that are not possible in the original problem, such as $a\ D\ (B\ d)\ (F\ (B\ (B\ D)))$ which results from rewriting the leftmost occurrence of y_0 to D and the rightmost occurrence to $B\ D$. The actual parameters D and $B\ y_0$ have become confused because, according to the current context C_0, they have the same acceptance and rejection types. In this way, *types direct the abstraction* and refinement will occur because type information in the context increases.

By contrast, the vertex $(B\ d,\, q_0)$ has an actual parameter whose acceptance and rejection types are non-trivial. The terminal d has acceptance type $\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma^0_\exists)(d) = q_1$ and rejection type $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma^0_\forall)(d) = q_0$. We have not yet created a new variable with those types, so we do so now, assigning the new variable y_1 the acceptance type q_1 and the rejection type q_0. We record that d is a possible instantiation of y_1 and label the successor $(B\ y_1,\, q_0)$. Following the same method, this new vertex has itself as successor. Officially we ought to note that y_1 is a possible instance of y_1, but such trivial circularities will make no difference to the outcome so we will omit it for brevity.

Let us now consider the frontier vertex $(y_0,\, q_0)$, which is variable headed. At variable headed configurations the consequences of the abstraction are felt. A vertex of the form $(y\ s_1 \cdots s_n,\, q)$ has a successor $(t\ s_1 \cdots s_n,\, q)$ for each binding $y \mapsto t$. For reasons that we have already discussed, its child $(B\ y_0,\, q_0)$ has itself as a successor. Its other child is the vertex $(D,\, q_0)$ whose contraction is $(d,\, q_0)$, however, because $\Gamma^0_\forall \vdash_{\mathcal{A}^c} d : q_0$ we do not add vertex $(d,\, q_0)$ to the graph and so $(D,\, q_0)$ has no successors. We say that $(D,\, q_0)$ is a *rejecting leaf*. This completes the construction of the abstract configuration graph.

We now consider any rejecting leaves in the graph. Rejecting leaves are configurations $(F\,s_1 \cdots s_n,\, q)$ for which the correspond-

ing type assignment $F\ s_1 \cdots s_n : q$ is not provable in either $\Gamma_\exists$ nor $\Gamma_\forall$, yet the type assignment associated with the contraction is provable in $\Gamma_\forall$. This points directly to a weakness in the context. In our example, $\Gamma^0_\forall \vdash_{\mathcal{A}^c} d : q_0$ is provable, so d is a tree rejected by $\mathcal{A}^\perp$ and D reduces to d in one step, yet $\Gamma_\forall \vdash_{\mathcal{A}^c} D : q_0$ is not provable! The rejection environment $\Gamma_\forall$ *ought* to type $D : q_0$, since D generates a tree that is rejected by $\mathcal{A}^\perp$, but it doesn't. Hence, we have discovered a new rejection type assignment. The vertices that we classify as representing typing assignments that ought to have been provable under the rejecting environment are collectively called the *rejecting region* and from this region new rejection types are extracted. Where the rejecting region is, roughly speaking, all the vertices that "definitely" can reach rejecting leaves, the *accepting region* is all the vertices that "definitely" cannot reach rejecting leaves. New acceptance types are extracted from the accepting region. In this case, the rejecting region is the single vertex $(D,\ q_0)$ and the accepting region is all the B headed configurations.

Type extraction from the regions follows a similar approach to that defined in [14]. Briefly, a type is assigned to each prefix s of the term component of each vertex $(s\ t_1 \cdots t_n,\ q)$ in the accepting region. The type assigned to s is constructed by recursively computing the type σ assigned to t_1 by considering all the ways in which t_1 is used within the region (the vertices where t_1 is itself a prefix), recursively computing the type τ assigned to the prefix $s\,t_1$ and then forging the arrow $\sigma \to \tau$. The base case is where the entire term component of the pair, considered as a trivial prefix of itself, is assigned the type q. The rejecting region is handled similarly, but care must be taken to only consider uses of t_1 in vertices that are strictly closer to the leaves so that a well founded co-consistency argument can be given in the end. The type assignments to prefixes that are themselves non-terminal symbols are extracted and added to the appropriate environments. In this case, there is only one vertex in the rejecting region, so its only prefix is assigned the type $D : q_0$. There are three vertices in the accepting region, of which the prefix B of $(B\ d,\ q_0)$ is assigned acceptance type $q_1 \to q_0$ since d is known to have type q_0, prefix B of $(B\ y_0)$ is assigned type $\top \to q_0$ since there are no uses of y_0 in the region and, for the same reason, prefix B of $(B\ y_1)$ is assigned the type $\top \to q_0$. The new context is therefore $C_1 = \langle\{B : (\top \to q_0) \wedge (q_1 \to q_0)\},\ \{D : q_0\}\rangle$.

Since the new context neither types $S : q_0$ as accepting nor as rejecting, the process repeats, but this time with more type information, so more of the parameters to calls will be distinguished. In the second iteration the rejecting region is the entire graph. Consequently, $S : q_0$ is added to the rejecting environment and the algorithm terminates, correctly deducing that the input is a no-instance.

4. An abstraction refinement algorithm

We now present the algorithm formally. We first give the definition of the key construction, the abstract configuration graph. We then describe how from the graph one can carve out the accepting and rejecting regions and the notion of type extraction appropriate to each. Finally we show how graph construction, regioning and type extraction come together to form a single iteration and we present the algorithm as the repetition of this process.

Assumption. *For the rest of this section, assume a recursion scheme $\mathcal{G} = \langle\Sigma, \mathcal{N}, \mathcal{R}, S\rangle$ and an alternating trivial automaton $\mathcal{A} = \langle\Sigma, Q, \delta, q_0\rangle$. By an abuse, we will refer to the type system induced by $\mathcal{A}$ simply as $\mathcal{A}$ and the type system induced by $\mathcal{A}^c$ simply as $\mathcal{A}^c$.*

4.1 Construction of abstraction

The abstraction is based on a traditional CFA [10], in the sense of over-approximating reduction using an abstract environment. An important twist on the usual formulation is that here every variable in the environment is associated with a kind and a pair of intersection types.

Typed variables. The main mechanism for abstraction will be a set of typed variables. By means of an abstract environment (to be described shortly) each variable represents the set of terms that can be obtained from it by repeated substitution. Each variable has three pieces of associated type information: an acceptance type, a rejection type and a kind. An essential part of the algorithm is in ensuring that type information is invariant across the abstraction, i.e. if a variable abstracts a set of terms, then every term in the set shares the same acceptance type, rejection type and kind as the variable (according to the current context).

Definition 1. *Let us say that a kind κ is an* argument kind *just if there is some binding $F : \kappa_1 \to \cdots \to \kappa_n \to o \in \mathcal{N}$ and some i such that $\kappa = \kappa_i$. Let* var *be a bijection, mapping the finite set of all triples of the form $(\sigma_\mathsf{A},\ \sigma_\mathsf{R},\ \kappa)$ consisting of kinded types $\sigma_\mathsf{A} :: \kappa$ and $\sigma_\mathsf{R} :: \kappa$,where κ is an argument kind, to a finite set of term variables $\mathcal{Y} \subseteq \mathcal{V}$. Given such a variable $y \in \mathcal{Y}$, we will write $\mathsf{A}(y)$ for the first component of $\mathsf{var}^{-1}(y)$, $\mathsf{R}(y)$ for the second and $\mathsf{K}(y)$ for the third.*

Type context. The algorithm is ultimately concerned with constructing a pair of type environments $\langle\Gamma_\exists, \Gamma_\forall\rangle$ such that $\Gamma_\exists$ is $(\mathcal{G},\mathcal{A})$-consistent and $\Gamma_\forall$ $(\mathcal{G},\mathcal{A}^c)$-co-consistent. We will speak of $\Gamma_\exists$ as the "acceptance" type environment and $\Gamma_\forall$ as the "rejection" type environment. Furthermore, we stipulate that every such pair of environments, which we shall call a *type context*, understands the basic assumptions we have made about the typed variables, i.e. the type information contained in A and R (as defined in Definition 1, but viewed as type environments for the typed variables) is also contained in $\Gamma_\exists$ and $\Gamma_\forall$ respectively.

Definition 2. *A* type context *$C = \langle\Gamma_\exists,\ \Gamma_\forall\rangle$ is a pair of intersection type environments for which the following conditions hold:*

(i) $\Gamma_\exists :: \mathcal{N} \cup \mathsf{K}$
(ii) $\Gamma_\forall :: \mathcal{N} \cup \mathsf{K}$
(iii) For all $y \in \mathcal{Y}$, $\Gamma_\exists(y) = \mathsf{A}(y)$ and $\Gamma_\forall(y) = \mathsf{R}(y)$

Abstract configurations. As mentioned in the previous section, the abstraction itself is a finite representation of the possibly infinite *configuration graph*, as defined by [14]. In this *concrete* configuration graph, the configurations are pairs of a *closed* term (a reduct of the start symbol of the scheme) and a state of the automaton, and the edges that connect them must respect the constraints of both the reduction relation of the scheme and of the transition function of the automaton. A configuration $(t,\ q)$ can be read as an assertion: the tree generated by t is accepted by $\mathcal{A}^\perp$ from state q. In the *abstract* configuration graph, defined shortly, configurations are still pairs of term and state, but now the term is abstract, which in our setting means that it can contain free occurrences of typed variables.

Definition 3. *An abstract configuration is a pair $(t,\ q)$ in which $\mathcal{N} \cup \{y : \mathsf{K}(y) \mid y \in \mathsf{FV}(t)\} \vdash t : o$ is a term and $q \in Q$ is a state. We say that a term s is a* prefix *of a term t just if t has the form $s\ t_1 \cdots t_n$ for some $n \in \mathbb{N}$. A* configuration prefix *is a pair $(c,\ s)$ in which c is a configuration of shape $(t,\ q)$ and s is a prefix of t.*

Abstract typability. The central idea of the algorithm is that the type bindings contained in the context constitute a concise summary of all the information that has been gathered about the scheme and its reducts, as far as acceptance by the property automaton is concerned. We will use the type context to judge whether the assertions represented by configurations are true or not, based on the following simple notion of typability.

Definition 4. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and let (t, q) be an abstract configuration. We say that (t, q) is C-*accepted *just if $\Gamma_\exists \vdash_{\mathcal{A}} t : q$. We say that (t, q) is C-*rejected *just if $\Gamma_\forall \vdash_{\mathcal{A}^c} t : q$. We say that (t, q) is C-*unknown *just if it is neither C-accepted nor C-rejected.*

Until the very last iteration of the algorithm, the configuration (S, q_0), which is the root of the abstract configuration graph, will be C-unknown to all the associated contexts C, but after the last iteration enough type information will have been contributed to the final context C' in order that (S, q_0) will be seen to be either C'-accepting or C'-rejecting.

Abstract configuration graph. The vertices of the abstract configuration graph are either abstract configurations or finite sets of abstract configurations. Viewed as an assertion, a vertex which is a finite set of configurations $\{(s_1, q_1), \ldots, (s_n, q_n)\}$ should be interpreted *conjunctively*, i.e. as requiring that for each $i \in [1..n]$, s_i generates a tree that is accepted from state q_i.

Definition 5. *An* abstract configuration graph *A is a tuple $\langle V, E, B \rangle$ in which $\langle V, E \rangle$ is a directed graph and B is a set of mappings from variables $y \in \mathcal{Y}$ to terms $t \in T_\Sigma(\mathcal{Y}, \mathcal{N})$. Each vertex $v \in V$ is either (i) an abstract configuration or (ii) a finite set of abstract configurations; and edges $E \subseteq V \times V$ are unlabelled. Given a typing context C, the* abstract configuration graph of *C, denoted $\mathsf{ACG}(C)$, is the abstract configuration graph $\langle V_C, E_C, B_C \rangle$ defined inductively by the system in Figure 4.*

The set B of *bindings* acts as the abstract environment for the purposes of defining the abstraction. We consider motivation of each of the rules of the inductive definition in turn. First, the rule (G1) defines the root of the graph. The premise ensures that, if we already know that S generates a tree that is either accepted from q_0 or rejected from q_0 then we need not do any state space exploration. This kind of premise is common to many of the rules to ensure that work is not done unnecessarily. In fact, one can state an invariant about the abstract typability of the vertices in any abstract configuration graph:

Lemma 1. *Let C be a context. For each configuration $c \in V_C$, c is C-unknown.*

In case (S, q_0) were C-accepting or C-rejecting, the graph would be empty and the sequence of contexts will stabilise.

Rule (G2) simulates the contraction of a redex, but it does so in an abstract way. To apply the rule requires that a configuration $(F\ s_1 \cdots s_n, q)$ containing a redex occurs in the graph. The consequence is that an abstraction of the contraction of that redex is added as a new configuration. However, it is abstract because, rather than substituting actual parameters for formals, typed variables are substituted for the formals. These typed variables must be appropriate for the actuals that they abstract, hence there is the constraint that, if y_i abstracts actual parameter s_i, then it had better be that $y_i = \mathsf{var}(\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i), \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i), \mathsf{K}(s_i))$. This ensures that type information is invariant across the abstraction, in the following sense:

Proposition 1. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. For all $y \mapsto t \in B_C$, $\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(y) = \bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(t)$ and $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(y) = \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(t)$.*

To properly define the abstraction in terms of the new variable y_i, a binding is added to B_C with the effect that $y_i \mapsto s_i$. Consequently, we may think that s_i is in the set of terms abstracted by y_i.

Rule (G3) simulates a transition of the automaton on reading a terminal symbol: if there is a terminal symbol-headed configuration $(a\ s_1 \cdots s_n, q)$ in the graph, then its children comprise all of the possible satisfying assignments to $\delta(q, a)$ expressed as sets of configurations. Recalling that each vertex that is a set of configurations should be thought of conjunctively, the children of $(a\ s_1 \cdots s_n, q)$, taken as a whole, should be thought of disjunctively – $a\ s_1 \cdots s_n$ generates a tree accepted from state q just if all the configurations contained in some child (satisfying assignment) are shown to be accepted. Rule (G4) simply decomposes set vertices into their constituent configurations. Therefore, the children of a set vertex should be thought of conjunctively.

(G1) Whenever all the following are true:
- (i) (S, q_0) is C-unknown

then all the following are also true:
- $(S, q_0) \in V_C$

(G2) Whenever all the following are true:
- (i) $(F\ s_1 \cdots s_n, q) \in V_C$
- (ii) $\mathcal{R}(F) = \lambda x_1 \cdots x_n.\, t$
- (iii) $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_n \to o$
- (iv) $(t[y_1/x_1, \cdots, y_n/x_n], q)$ is C-unknown
- (v) for each $i \in [1..n]$, $y_i = \mathsf{var}(\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i), \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i), \kappa_i)$

then all the following are also true:
- $(t[y_1/x_1, \cdots, y_n/x_n], q) \in V_C$
- $\langle (F\, s_1 \cdots s_n, q), (t[y_1/x_1, \cdots, y_n/x_n], q) \rangle \in E_C$
- for each $i \in [1..n]$: $y_i \mapsto s_i \in B_C$

(G3) Whenever all the following are true:
- (i) $(a\ s_1 \cdots s_n, q) \in V_C$
- (ii) $S \models \delta(q, a)$
- (iii) for all $(i, q') \in S$, (s_i, q') is not C-rejected

then all the following are also true:
- $\{(s_i, q') \mid (i, q') \in S\} \in V_C$
- $\langle (a\ s_1 \cdots s_n, q), \{(s_i, q') \mid (i, q') \in S\} \rangle \in E_C$.

(G4) Whenever all the following are true:
- (i) $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in V_C$
- (ii) $i \in [1..n]$
- (iii) (s_i, q_i) is not C-accepted

then all the following are also true:
- $(s_i, q_i) \in V_C$
- $\langle \{(s_1, q_1), \ldots, (s_n, q_n)\}, (s_i, q_i) \rangle \in E_C$

(G5) Whenever all the following are true:
- (i) $(y\ s_1 \cdots s_n, q) \in V_C$
- (ii) $y \mapsto t \in B_C$

then all the following are also true:
- $(t\ s_1 \cdots s_n, q) \in V_C$
- $\langle (y\ s_1 \cdots s_n, q), (t\ s_1 \cdots s_n, q) \rangle \in E_C$

Figure 4. Abstract configuration graph construction.

Finally, rule (G5) ties the knot on the abstraction by considering the case when a typed variable is in head position in a configuration. In this case, the binding set is consulted and a node is added for each binding to the appropriate variable. We will think of the children of such a vertex conjunctively: for $y\ s_1 \cdots s_n$ to generate a tree accepted from state q, it had better be that every term that it abstracts generates a tree accepted from state q.

Due to the abstraction at the point of contraction in (G2) and the limited substitution (only in head position) in (G5), $\mathsf{ACG}(C)$ is necessarily a finite construction. In fact, we can go further:

Lemma 2. *Let C be a type context. Then the size of V_C is bounded by a polynomial function of the size of the scheme.*

Classification of leaves. Let us consider for a moment the leaves of $\mathsf{ACG}(C)$ for some type context C. It follows from the definition that the leaves all have a particular form. Every leaf is a configuration headed by a non-terminal symbol, i.e. a redex. Moreover, each such redex, if contracted using rule (G2), would yield a new configuration which is already known to be either C-accepting or C-rejecting. It is for this reason that such configurations are leaves: (G2) does not apply because the fourth premise would be violated.

Definition 6. *Given a type context $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$, the leaves (i.e. those vertices that have no children) of $\mathsf{ACG}(C)$ can be classified into two sets:*

(ACCEPTING LEAVES) *These leaves are configurations of the form $(F\ s_1 \cdots s_n, q)$ where $\mathcal{R}(F) = \lambda x_1 \ldots x_n.t$, for each $i \in [1..n]$, there is a typed variable y_i such that $\mathsf{A}(y_i) = \bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i)$ and $\Gamma_\exists \vdash_{\mathcal{A}} t[y_1/x_1, \ldots, y_n/x_n] : q$.*

(REJECTING LEAVES) *These leaves are configurations of the form $(F\ s_1 \cdots s_n, q)$ where $\mathcal{R}(F) = \lambda x_1 \ldots x_n.t$, for each $i \in [1..n]$, there is a typed variable y_i such that $\mathsf{R}(y_i) = \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i)$ and $\Gamma_\forall \vdash_{\mathcal{A}^c} t[y_1/x_1, \ldots, y_n/x_n] : q$.*

Note that a rejecting leaf is not itself C-rejecting, by Lemma 1 since it is in the graph it is necessarily C-unknown, but its contractum is C-rejecting. Similarly accepting leaves are not themselves C-accepting, but the contractum of an accepting leaf is C-accepting.

Lemma 3. *Let C be a context. Every leaf in $\mathsf{ACG}(C)$ is accepting or rejecting.*

4.2 The rejecting region

Region of rejection. The construction of an ACG from a given type context C is a method for analysing the type context. By constructing the graph it is possible to see where the information in the type context is deficient, and the main tools for identifying and correcting deficiencies are the regions and region type extraction respectively. Consider a rejecting leaf v of the form $(F\, s_1 \cdots s_n, q)$. By definition, the contraction of this configuration using (G2) would yield a configuration $(t[y_1/x_1, \ldots, y_n/x_n], q)$ which is already C-rejecting. In other words, the tree generated by *any* term of the form $t[t_1/y_1, \ldots, t_n/y_n]$ such that $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(t_i) \leq \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(y_i)$ for each i is sure to be rejected from state q. Assuming that $\Gamma_\forall$ is co-consistent, this follows because necessarily $\Gamma_\forall \vdash_{\mathcal{A}^c} t[t_1/y_1, \ldots, t_n/y_n] : q$. Recalling Proposition 1, one such sequence of t_i are the actual parameters of the term component of the rejecting leaf we started with: v. Hence, because we know that the contractum of the term part of v generates a tree that is rejected from state q_0, necessarily the term part of v itself generates a tree that is rejected from state q_0. So we have identified that v should be classified as C-rejecting (but is not currently). Through analogous reasoning (and remembering the conjunctive and disjunctive interpretations of the child relation in the graph), it is possible to identify other such vertices which are necessarily rejecting. The collection of all such is called the rejecting region.

Definition 7. *Given a context C, we define a subset $\mathsf{RR}(C) \subseteq V_C$ of the vertices of $\mathsf{ACG}(C)$, called the* rejecting region*, inductively:*

(R1) If c is a rejecting leaf then $c \in \mathsf{RR}(C)$.
(R2) If $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in V_C$ and there exists $j \in [1..n]$ and $(s_j, q_j) \in \mathsf{RR}(C)$ then $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in \mathsf{RR}(C)$.
(R3) If $\langle (F\ s_1 \cdots s_n, q), (t, q) \rangle \in E_C$ and $(t, q) \in \mathsf{RR}(C)$ then $(F\ s_1 \cdots s_n, q) \in \mathsf{RR}(C)$.
(R4) If $(a s_1 \cdots s_n, q) \in V_C$ and, for every v, $\langle (a s_1 \cdots s_n, q), v \rangle \in E_C$ implies $v \in \mathsf{RR}(C)$, then $(a\ s_1 \cdots s_n, q) \in \mathsf{RR}(C)$.
(R5) If $(y\ s_1 \cdots s_n, q) \in V_C$ and, for all $y \mapsto t \in B_C$, $(t\ s_1 \cdots s_n, q) \in \mathsf{RR}(C)$ then $(y\ s_1 \cdots s_n, q) \in \mathsf{RR}(C)$.

Unless it is the final iteration of the algorithm, the rejecting region will always be non-empty. The fact that an absence of rejecting leaves is an absence of counterexamples in the abstraction is formalised later, in Lemma 6.

Rejection type extraction. The vertices in the rejecting region are those configurations, that we have identified by constructing $\mathsf{ACG}(C)$, which *should* be classified by the context as rejecting, but *are not* – each is necessarily C-unknown, since it belongs to the graph. So the rejecting region represents a weakness in the context. To remedy it, from the region we will extract new type information to be added to the context ready for the next iteration.

Definition 8. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a typing context and $v \in \mathsf{RR}(C)$. A witness to the membership of v in $\mathsf{RR}(C)$ is a proof tree T rooted at the statement $v \in \mathsf{RR}(C)$ and constructed according to the rules (R1) – (R5). We describe an assignment of type environments $\mathcal{M}(T)$ to proof trees T, inductively on the shape of the proof.*

(M1) If the proof is by (R1) then v is a configuration of the form $(F\ s_1 \cdots s_n, q)$, and we set $\mathcal{M}(T)$ to be the single binding:

$$F : \bigwedge \mathbb{T}(\Gamma_\forall)(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall)(s_n) \to q$$

(M2) If the proof is by (R2) then v is a set $\{(s_1, q_1), \ldots, (s_n, q_n)\}$ and for some $j \in [1..n]$ there is an immediate sub-proof T' of (s_j, q_j). We set $\mathcal{M}(T) = \mathcal{M}(T')$.

(M3) If the proof is by (R3) then v is a configuration of the form $(F\ s_1 \cdots s_n, q)$ and, necessarily, there is an immediate sub-proof T' of (t, q). We take for $\mathcal{M}(T)$ the environment $\mathcal{M}(T')$ augmented by the binding:

$$F : \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_n) \to q$$

(M4) If the proof is by (R4) then v is of the form $(a\ s_1 \cdots s_n, q)$ with a set W of children. For each $w \in W$, there is a sub-proof T_w. We set $\mathcal{M}(T) = \biguplus \{\mathcal{M}(T_w) \mid w \in W\}$.

(M5) If the proof is by (R5) then v is a configuration of the form $(y\ s_1 \cdots s_n, q)$ and, necessarily, for each $y \mapsto t \in B_C$ there is an immediate sub-proof T_t of $(t\ s_1 \cdots s_n, q)$. Let us write $\mathcal{M}(T_y)$ simply as notation for the environment given by $\biguplus \{\mathcal{M}(T_t) \mid y \mapsto t \in B_C\}$. We take for $\mathcal{M}(T)$ the environment:

$$y : \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T_y))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T_y))(s_n) \to q$$

Finally, we define a type environment, $\mathsf{env}_\mathsf{R}(C)$, whose domain is a subset of $\mathsf{dom}(\mathcal{N})$ and which is extracted from $\mathsf{RR}(C)$ by:

$$\mathsf{env}_\mathsf{R}(C)(F) = \bigwedge \{\mathcal{M}(T)(F) \mid \exists\, c \in \mathsf{RR}(C) \text{ with witness } T\}$$

So the types are extracted in an inductive fashion, starting from the leaves of each witness with the environment $\Gamma_\forall$ and working back-

wards, adding new types along with way. It is this well-foundedness that ensures that the types that are extracted are all "correct":

Lemma 4. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. If $\Gamma_\forall$ is $\mathcal{G}$-co-consistent in $\mathcal{A}^c$ then $\Gamma_\forall \uplus \mathsf{env}_\mathsf{R}(C)$ is $\mathcal{G}$-co-consistent in $\mathcal{A}^c$.*

Furthermore, whenever the rejecting region is non-empty, then genuinely new type information will be extracted. Taken together with Lemma 6, the following result is the key measure of progress in the algorithm.

Lemma 5. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and $\mathsf{ACG}(C)$ have some rejecting leaf. Then $\mathsf{env}_\mathsf{R}(C) \setminus \Gamma_\forall \neq \emptyset$.*

4.3 The accepting region

Region of acceptance. In a similar way, the accepting region serves to identify those configurations that *should* be classified as accepting by the type context, but which are not. The rules by which vertices can be inferred to be accepting are all complimentary to those that define the rejecting region (except for the case of variable-headed nodes, which are conjunctive in both regions) and, indeed, the construction is coinductive.

Definition 9. *Given a typing context C, we define a subset $\mathsf{RA}(C) \subseteq V_C$ of the vertices of $\mathsf{ACG}(C)$, called the* accepting region*, coinductively by:*

(A1) If $(F\ s_1 \cdots s_n,\ q) \in \mathsf{RA}(C)$ with successor $(t,\ q) \in V_C$, then $(t,\ q) \in \mathsf{RA}(C)$.
(A2) If $(a\ s_1 \cdots s_n,\ q) \in \mathsf{RA}(C)$, then there is some S such that $S \models \delta(q,\ a)$ and $\{(s_i,\ q') \mid (i,\ q') \in S\} \in \mathsf{RA}(C)$.
(A3) If $\{(s_1,\ q_1), \ldots, (s_m,\ q_m)\} \in \mathsf{RA}(C)$ then, for all $i \in [1..m]$, $(s_m,\ q_m) \in \mathsf{RA}(C)$.
(A4) If $(y\ s_1 \cdots s_n,\ q) \in \mathsf{RA}(C)$ then, for all $y \mapsto t \in B_C$, $(t\ s_1 \cdots s_n,\ q) \in \mathsf{RA}(C)$.
(A5) If $c \in \mathsf{RA}(C)$ is a leaf in V_C then c is an accepting leaf.

However, unlike the case for rejecting region there is no similar guarantee of non-emptiness on non-final iterations. It is perfectly possible that on any given iteration, the accepting region may be empty. In contrast, the absence of rejecting leaves, and hence emptiness of the rejecting region, leads to termination.

Lemma 6. *Let C be a type context and $\mathsf{ACG}(C)$ have no rejecting leaves. Then $\mathsf{ACG}(C) = \mathsf{RA}(C)$.*

Thus, in particular, $\mathsf{RA}(C)$ will contain the root and so $S : q_0$ will be added to the accepting environment, signalling termination.

Acceptance type extraction. To extract new type information from the accepting region we follow the approach of Kobayashi and Ong [12, 14], in which types are assigned to prefixes of configurations recursively based on the kind of the prefix.

Definition 10. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. To each prefix $(c,\ s)$ of each configuration $c \in \mathsf{RA}(C)$, we assign a strict type $\mathsf{extr}(c,\ s)$, which is defined inductively over the structure of the kind of s.*

(i) If s is of base kind, necessarily c is of the form $(s,\ q)$ and set $\mathsf{extr}(c,\ s) = q$.
(ii) If s is of arrow kind, necessarily c is of the form $(s\,t_1 \cdots t_n,\ q)$. Let W be the set of accepting region configurations with prefix t_1. Set:

$$\mathsf{extr}(c,\ s) = \bigwedge \mathbb{T}_\mathcal{A}(\Gamma_\exists)(t_1) \wedge \bigwedge_{c' \in W} \mathsf{extr}(c',\ t_1) \to \mathsf{extr}(c,\ s\,t_1)$$

We define a type environment, $\mathsf{env}_\mathsf{A}(C)$, whose domain is a subset of $\mathsf{dom}(\mathcal{N})$ and which is extracted from $\mathsf{RA}(C)$ by:

$$\mathsf{env}_\mathsf{A}(C)(F) = \bigwedge \{\tau \mid \exists\, c \in \mathsf{RA}(C) \cdot \mathsf{extr}(c,\ F) = \tau\}$$

The acceptance types extracted in this way are all "correct":

Lemma 7. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context. If $\Gamma_\exists$ is $(\mathcal{G},\mathcal{A})$-consistent then also $\Gamma_\exists \uplus \mathsf{env}_\mathsf{A}(C)$ is $(\mathcal{G},\mathcal{A})$-consistent.*

4.4 Fixed point construction

Abstraction refinement. Finally, we are in a position to describe the overall abstraction refinement loop. Starting from a context C_0 that contains only the type assumptions on typed variables used by the abstraction, on each iteration the algorithm analyses the given context, say C_i, by constructing $\mathsf{ACG}(C_i)$; it then identifies deficiencies in C_i by constructing regions and attempts to repair those deficiencies by extracting new environments. Eventually, the type $S : q_0$ will be extracted from one of the regions and the algorithm will terminate.

Definition 11. *Recall A and R in Definition 1. The algorithm consists of constructing an eventually stable sequence of type contexts $(C_i)_{i \in \mathbb{N}}$ as follows:*

$$\begin{array}{lll} C_0 & = \langle \Gamma_\exists^0,\ \Gamma_\forall^0 \rangle & = \langle \mathsf{A},\ \mathsf{R} \rangle \\ C_{k+1} & = \langle \Gamma_\exists^{k+1},\ \Gamma_\forall^{k+1} \rangle & = \langle \Gamma_\exists^k \uplus \mathsf{env}_\mathsf{A}(C_k),\ \Gamma_\forall^k \uplus \mathsf{env}_\mathsf{R}(C_k) \rangle \end{array}$$

with limit, say $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$. Then if $q_0 \in \Gamma_\exists(S)$ answer YES *and otherwise answer* NO.

Since the initial environments $\Gamma_\exists^0$ and $\Gamma_\forall^0$ are trivially $\mathcal{G}$-consistent in $\mathcal{A}$ and $\mathcal{G}$-co-consistent in $\mathcal{A}^c$ respectively and since every extension of these environments by env_A and env_R preserves this property, it follows that the limit of the sequence also enjoys the property and hence can be relied upon to decide the model checking problem. Furthermore, since progress is guaranteed by Lemma 6 and Lemma 5, and the size of rejecting environment $\Gamma_\forall$ is bounded by the number of well-kinded types, we can state the following correctness theorem:

Theorem 2. *For any $\mathcal{G}$, $\mathcal{A}$, the algorithm terminates and:*

- *Answer* YES *implies $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$.*
- *Answer* NO *implies $\mathsf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^\perp)$.*

Furthermore, since each ACG is, in the worst case, polynomial in the size of the scheme (but in general, hyper-exponential in the order of the scheme) and the amount of work involved in computing the ACG, the regions and type extraction is polynomial in the size of the scheme, it follows that each iteration of the algorithm takes, in the worst case, an amount of time polynomial in the size of the scheme. Since the number of iterations is bounded by the number of well-kinded types, which is also polynomial in the size of the scheme, it follows that the algorithm as a whole is polynomial in the size of the scheme, assuming its order and arity and the size of the automaton are taken to be fixed.

5. Implementation and evaluation

We have implemented the algorithm in a prototype tool, called PREFACE, which is written in F# and available to download from http://mjolnir.cs.ox.ac.uk/web/preface.

5.1 Implementation

To ensure efficiency we have taken a number of decisions about how to code specific aspects of the algorithm which deviate from the presentation. Rather than constructing them as part of initialisation, we build the maps A and R lazily, adding bindings as they are needed by applications of rule (G2). Further, the implementation uses a flow analysis with increased accuracy, distinguishing between instances of arguments using not just the triple of acceptance type, rejection type and kind, but additionally the formal parameter

Benchmark	Rules	Ord	Dec	PREFACE	HORSAT	HORSATT	C-SHORE	GTRECS2	TRAVMC	TRECS
cfa-psdes	237	7	A	0.51	0.28	1.81	3.44	–	–	–
cfa-matrix-1	383	8	A	0.61	0.73	6.30	18.58	–	–	–
cfa-life2	898	14	A	1.46	5.94	–	–	–	–	–

Table 2. Benchmarks of category 2.

Benchmark	Rules	Ord	Dec	PREFACE	HORSAT	HORSATT	C-SHORE	GTRECS2	TRAVMC	TRECS
exp2-1600	1606	2	A	8.39	–	–	–	10.47	–	–
exp2-3200	3206	2	A	17.51	–	–	–	59.13	–	–
exp2-6400	6406	2	A	39.58	–	–	–	–	–	–
exp2-12800	12806	2	A	92.19	–	–	–	–	–	–
exp4-400	408	4	A	14.12	–	106.53	–	–	–	–
exp4-800	808	4	A	30.55	–	–	–	–	–	–
exp4-1600	1608	4	A	71.06	–	–	–	–	–	–
exp4-3200	3208	4	A	–	–	–	–	–	–	–

Table 3. Benchmarks of category 3.

Benchmark	Rules	Ord	Dec	PREFACE	TRECS
map_filter-e	64	5	R	0.53	0.01
fold_left	65	4	A	0.39	0.03
fold_right	65	4	A	0.39	0.03
forall_eq_pair	66	4	A	0.39	0.03
forall_leq	66	4	A	0.39	0.03
a-cppr	74	3	R	0.38	0.01
search-e	96	5	R	0.90	0.01
search	119	4	A	0.46	1.04
map_filter	143	5	A	0.51	0.13
risers	148	5	A	0.44	0.33
r-file	156	2	A	0.82	1.50
fold_fun_list	197	6	A	0.44	0.89
zip	210	3	A	0.58	15.10

Table 1. Benchmarks of category 1.

and the state component of the calling configuration. Since intersection type checking is frequently invoked as part of the decision procedure, we aim to ensure it is done as efficiently as possible. Hence, we omit subtype checking[3] and hash cons the intersection types. Finally, rather than compute all possible witnessing trees for any given vertex $v \in \mathsf{RR}(C)$, we take one representative, which is a function of the construction of the region.

5.2 Evaluation

We have evaluated the tool on the large collection of recursion scheme model checking instances found online and in the related literature. The full listing of results is available in the appendix of the long version of this paper, here we aim to present a very small representative sample in order to describe the general trends.

We have picked a number of benchmarks from three categories, which are displayed in Tables 1, 2 and 3 respectively. The benchmarks were run on an Intel Xeon machine with 12GB of RAM and 4 cores running at 2.4GHz, limiting the run-time of each tool on each benchmark to 2 minutes. In all cases the columns are, respectively, the name of the benchmark, the number of rules (equations), the order of the scheme, whether the tree is accepted (A) or rejected (R) by the property automaton. The remaining columns list the time taken by each tool from start to finish, which is either given in seconds, or marked "–" in case the tool ran out of resources.

[3] Note that this does not affect the soundness or completeness of the decision procedure.

Category 1. The first category consists of model checking instances that have arisen from OCaml verification problems via the predicate abstraction tool MoCHi [25]. Although MoCHi can solve many complex examples, the scalability of a full blown predicate abstraction tool for higher-order programs is still an open topic of research. Consequently the problem instances derived from this tool are exclusively quite small and mostly less than 100 rules. MoCHi generates a mild extension of recursion schemes called RSFD [16], which are currently only supported by PREFACE and TRECS. Our tool PREFACE can typically solve each of these instances in less than 0.5 seconds, but this is already roughly an order of magnitude slower than TRECS. However, the overhead incurred by JIT compilation on Mono is a major factor; when compiled ahead of time on Windows, the time taken by PREFACE to solve these instances is typically less than 0.05 seconds, although usually still slower than TRECS. As the benchmarks become slightly larger, towards the bottom of the table, the time taken by TRECS starts to lag behind the time taken by PREFACE, which is the start of a general trend in the data to follow.

Category 2. The second category consists of instances arising from a tool for performing exact flow analysis [26]. These examples are significantly larger than those of Category 1 and, indeed, form some of the largest instances on which HORS model checkers have been evaluated in the literature as of the time of writing. Although they have fewer than 1000 rules each, due to the nature of the verification algorithm that produces them, they have high order and very high maximum arity, with `cfa-life2` being order 14 with arity 29 functions. Consequently, many of the tools have difficulty, but among those that are able to solve these instances, the trend observed in the Category 1 examples can be seen to continue.

Category 3. The final category consists of instances of a family of schemes due to Kobayashi [13]. This family of instances was designed to be deliberately difficult for bounded model checking style algorithms such as the hybrid algorithm of TRECS, whilst simultaneously being a good indicator of the scalability of Kobayashi's linear time algorithm as implemented in GTRECS. Although these schemes are not "real" in the sense of arising from program analysis problems they seem a good measure of scalability since they generate hyper-exponentially sized trees (and hence use the full power of higher-order schemes), their certificates are proportional to the number of rules and they can push the model checkers much further since the family contains much larger schemes than can be produced by current verification tools. The first half of the table shows instances which are generated with order-2 schemes and the second half shows instances with order-4 schemes. The size of the

schemes roughly doubles within each half by row. As expected, GTRecS does a good job at solving even relatively large examples at order-2, though it has some difficulty at higher-orders. PREFACE does even better and, in contrast to the other tools, can be seen to solve these examples in time which is roughly linear in the number of rules.

5.3 Analysis

The good performance of our algorithm at scale must be attributed to the abstraction refinement approach that we have adopted. Since recursion schemes cannot destruct the trees that they create, their interesting behaviours are exclusively due to control flow arising from complex uses of higher-order functions. Hence, a CFA-style abstraction in combination with a property directed refinement works well, since this abstraction is particularly well suited to emphasising some of the essential structure of higher-order control flow and the refinement ensures that particular limitations of the CFA with respect to specific problem instances can be compensated for. A good example of this is in the Category 3 examples, whose very regular structure is determined by the analysis quickly and hence all are solved in exactly 3 iterations (independent of the number of rules or the order of the scheme).

However, although the refinement will eventually compensate for these particular limitations of the CFA, it is possible to construct instances in which the number of iterations required is unacceptably large with respect to the characteristics of the instance. Although such examples do not seem to occur in the corpus of instances drawn from the higher-order model checking literature and associated verification tools, we have been able to construct very small and simple schemes which exhibit this bad behaviour. The examples in Table 4 are based on a family of Boolean programs defined in [1]. Each of these instances is first order and consists of a few hundred rules, but the property automaton is strictly alternating. We record the number of iterations (Rnds) and the time (in seconds) taken by PREFACE and an extension, PREFACE$^+$, in the remaining columns.

		PREFACE		PREFACE$^+$	
Benchmark	Rules	Rnds	Time	Rnds	Time
t100	104	202	45.38	8	2.17
t200	204	402	178.81	8	3.99
t400	404	802	732.40	8	7.97
t800	804	1602	3074.03	9	18.50
t1600	1604	3202	13561.26	9	41.02

Table 4. Bad behaviour.

Each example `t`n consists of roughly n functions, which make exponentially many calls to each other in sequence, so that function $F1$ calls function $F2$ twice, function $F2$ calls function $F3$ twice and so on. However, what makes the examples expose bad behaviour in PREFACE is not the number of calls (which is hyper-exponentially smaller than the number of calls made by the `exp`n`-`m examples in Category 3), but the fact that each call is made once with a term that will eventually evaluate to true and once with a term that will eventually evaluate to false and that refutation of the property depends upon distinguishing between the two. On iteration $2i+2$, the flow analysis is only able to distinguish between the true and false variants of the calls made to functions Fj for $j \geq n - i$. Every 2 iterations, enough new information has been discovered in order to distinguish one more level of function calls, and hence each `t`n is solved after roughly $2n$ iterations.

This analysis suggests that not enough type information is being recovered from the ACG at each iteration and, indeed, by extending our implementation with heuristics for extracting more types, we have solved these examples more quickly. Our extension, labelled PREFACE$^+$ in the table, exploits the *incremental* nature of the algorithm. By this we mean the following characteristics:

(i) The algorithm makes progress on each iteration, in the sense of extracting new types (even if the number of types extracted is perhaps smaller than one would like).

(ii) On each iteration, the provenance of the context is not important, only the fact that it comprises environments that are consistent and co-consistent respectively.

(iii) The larger the context for a given iteration, the more accurate the analysis of that iteration.

Our extension consists of, in a separate thread running in parallel with the main algorithm, taking the ACG that has most recently been computed and, based on the relationships between the vertices, making informed "guesses" at possible new types. In general the guesses may be incorrect, in the sense of leading to the creation of an environment which is unjustifiable, so the new environments are first type-checked according to the rules of (co-)consistency. If they type-check, the new types are then added into the context at the earliest opportunity and the guessing process can be repeated.

This extension appears to work well to solve the examples in Table 4, cutting the time taken to process `t1600` down from almost 3 hours to under one minute! However, it seems unlikely to scale well to higher-orders, where the possible number of types from which to guess is much larger. Consequently, we do not consider this a satisfactory solution and leave to future work a proper treatment of this problem and that of the closely related area of how best to extract counter-example traces.

6. Related work

Higher-order model checking algorithms. Exemplified by the tool TRecS [11], the first *practical* algorithms model check HORS with respect to trivial automata, using intersection types as a finite representation of an infinite transition system. They start from the assignment of the automaton initial state to the start non-terminal S, and compute the "post-image" in the form of type information required to deduce unreachability of the error configurations. In TRecS, the types of non-terminals are first extracted from a finitely reachable part of the (infinite) configuration graph; the resultant type environment is then expanded and used as an over-approximation of a greatest fixpoint (of *shrink* [12]). PREFACE is similar to TRecS in that it implements a forward algorithm based on intersection types. However, in contrast, PREFACE extracts types from a finite, 0CFA-like abstraction of the configuration graph. Another major difference is that each iteration of PREFACE refines two type environments, one is potentially a certificate of automaton acceptance, and the other of automaton rejection.

Very fast for HORS of up to a few hundred rules, the runtime of TRecS is nonetheless hyper-exponential in the size of the input HORS. The first fixed-parameter polytime (in the size of HORS) algorithm is GTRecS [13] which consists simply of two fixpoint constructions. The key innovation is a game-semantic reading of the intersection types *qua* a pair of expansion relations, modelling the legals moves of the two players of an arena game. The tool TravMC [19] is also based on game semantics. The algorithm harvests *variable profiles*, but represented as intersection types, from the *traversals* [20] over the HORS being analysed.

Recently Broadbent et al. [4] have introduced an algorithm based on *collapsible pushdown automata* (CPDA) [8]—which are equi-expressive with HORS—and implemented in the tool C-SHORe [5]. Given an input co-trivial tree automaton, the algorithm uses a generalisation of the saturation algorithm for pushdown automata to compute the "pre-image" of the final error configurations, and checks if it includes the start state. Thus information

is propagated in the *backward* direction. Closely related are algorithms HORSAT and HORSATT [3]. Though based on saturation (of HORS rather than CPDA), they may be viewed as fixpoint computation of a function over type environments. To accelerate the pre-image computation, these algorithms benefit from a forward flow analysis, which excludes some irrelevant type bindings.

Abstraction refinement. Counterexample-guided abstraction refinement (CEGAR) was introduced by Clarke et al. [6] for symbolic model checking. The CEGAR loop was first applied to higher-order model checking by Ong and Ramsay [22] and by Kobayashi et al. [17]. The former addresses the undecidable problem of verifying safety properties of pattern-matching recursion schemes, using patterns to abstract properties. The latter is used in conjunction with predicate abstraction to verify simply-typed functional programs generated from infinite data domains such as integers. In contrast, PREFACE builds successively more accurate finite abstractions of the configuration graph of the HORS being analysed, from which potential certificates of acceptance and of rejection are derived.

Type-based flow analysis. Flow analyses were first applied to untyped languages. Jagannathan et al. [9] introduced a type-directed, polyvariant flow analysis for the predicative subset of System F, which can leverage types to analyse programs more precisely. Our algorithm uses an intersection type system for describing automaton definable properties and necessarily works in a situation in which not all type information is known; in contrast theirs uses more standard typing (System F) and starts from a situation in which all types are known. Thus their analysis is comparable to a single iteration of our algorithm where the associated context already contains all the possible correct type information. The property of *respecting types*, which is put forward by the authors as a measure of the appropriateness of a CFA for a typed language is, for us, actually an essential technical requirement in order to extract new, valid type information.

Plevyak and Chien [23] considered a constraint-based type inference for object-oriented programs. Like our algorithm, not all type information is known at the start, and they compute types iteratively based on a flow analysis. However, within a single iteration they do not distinguish based on type information, instead they distinguish based on clashes discovered in the previous iteration. A problem with their approach of distinguishing calls unconditionally is that there may be infinitely many counterexamples which are being distinguished one at a time. Because of possible nontermination, their method cannot handle recursion in general.

Acknowledgments

We are grateful to Naoki Kobayashi, Hiroshi Unno, Ryosuke Sato, Matthew Hague and Christopher Broadbent for providing their tools and contributing a large number of examples for benchmarking in Section 5. The first author was generously supported in this work by a graduate research grant from Merton College, Oxford.

References

[1] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'00*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[2] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[3] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *CSL'13*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl, 2013.

[4] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *ICALP'12*, volume 7392 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2012.

[5] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-SHORe: a collapsible approach to verifying higher-order programs. In *ICFP'13*, pages 13–24. ACM, 2013.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, pages 154–169. Springer-Verlag, 2000.

[7] M. Coppo and M. Dezani. An extension of the basic functionality theory for the lambda-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[8] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS'08*, pages 452–461. IEEE Computer Society, 2008.

[9] S. Jagannathan, S. Weeks, and A. K. Wright. Type-directed flow analysis for typed intermediate languages. In *SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1997.

[10] N. D. Jones. Flow analysis of lambda expressions. In *ICALP'81*, volume 115 of *LNCS*, pages 114–128. Springer, 1981.

[11] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL'09*, pages 416–428. ACM, 2009.

[12] N. Kobayashi. Model-checking higher-order functions. In *PPDP'09*, pages 25–36. ACM, 2009.

[13] N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FOSSACS 2011*, volume 6604 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2011.

[14] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS 2009*, pages 179–188. IEEE Computer Society, 2009.

[15] N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011.

[16] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL'10*, pages 495–508, 2010.

[17] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI'11*, pages 222–233. ACM, 2011.

[18] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Compututer Science*, 54(2–3):267–276, 1987.

[19] R. P. Neatherway, C.-H. L. Ong, and S. J. Ramsay. A traversal-based algorithm for higher-order model checking. In *ICFP'12*, pages 353–364. ACM, 2012.

[20] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS'06*, pages 81–90. IEEE Comp. Soc., 2006.

[21] C.-H. L. Ong. Models of higher-order computation: Recursion schemes and collapsible pushdown automata. In *LLRS'10*, pages 263–299. 2010.

[22] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*, pages 587–598. ACM, 2011.

[23] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA'94*, pages 324–340. ACM, 1994.

[24] S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. Long version: `http://mjolnir.cs.ox.ac.uk/papers/preface.pdf`.

[25] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *PEPM'13*, pages 53–62. ACM, 2013.

[26] Y. Tobita, T. Tsukada, and N. Kobayashi. Exact flow analysis by higher-order model checking. In *FLOPS'12*, volume 7294 of *LNCS*, pages 275–289. Springer, 2012.

[27] S. van Bakel. Strict intersection types for the lambda calculus. *ACM Computing Surveys*, 43(3):20, 2011.